

Comparing Web Server Architectures:

Serverless vs Traditional

Ege Cavusoglu

Dec 1st, 2021

Table of Contents

Introduction	4
Background	6
The Experiment	10
The Client	10
Web Servers	11
Database Schema	12
Results	13
Analysis	15
Scalability	15
Reliability	15
Performance	17
Cost	18
Conclusion	20
References	21

Executive Summary

Web servers are computers that are dedicated to handle requests sent over the internet.

Over the recent years, a new and open source framework called Serverless was developed to allow software engineers to deploy web applications that act as simple functions. This method has various infrastructural advantages over the traditional way of setting up web servers. This paper aims to analyze the optimal method of managing web applications for small scale businesses and startups by comparing Serverless and traditional web servers on scalability, reliability, performance, and costs.

To tackle the question, a simple experiment was set up to observe telemetry data on how serverless functions and traditional servers performed in a mocked scenario. Combined with the analysis of the working principles of the two methods and data reported from the cloud provider industry, we found out that Serverless provides a hassle-free developer experience, scalability, reliability and a pay-as-you-go cost model which makes it a perfect match for small scale businesses.

Introduction

Serverless is a new concept in cloud computing that allows software developers to deploy simple functions to the cloud without managing any infrastructure compared to the traditional method where they set up private servers (or virtual machines) to run their services. Just like any new technology, there are tradeoffs and advantages of using this new technology. Although traditional servers are theoretically more performant; when scaling, reducing costs and platform independency is part of the equation there are cases where Serverless becomes a very good option. This paper aims to analyze the optimal choice of server architecture for small scale projects and early stage startups considering various factors such as scalability, reliability, performance, and costs.

For the scope of this paper, a definition of the criteria must be made first. Let's examine the main ones that will be significant.

1. **Scalability**, according to VMware [1], “refers to the ability to increase or decrease IT resources as needed to meet changing demand.” In the context of web servers this means handling different loads of requests made to your application whether that is 100 requests/day or 1,000,000. Scalability is not something that can be mocked with a simple setup, but we will discuss how the two architectures propose solutions and compare their ease of use.
2. **Reliability** refers to the health of your services and their ability to complete & respond to the requests. It is possible that some computing resources might be experiencing down time, or various services fail to communicate successfully to complete a request which will hurt the reliability of the system. In our context, any request that returns an error (or does not return a response at all) is considered a faulty request. The experiment that is

setup will allow us to see a real life example of how the two architectures performed, and we will support this data with reliability reports from cloud providers aggregated from billions of daily requests.

3. **Performance** is the measure of how quickly web servers are able to respond to requests.

Performance might have various definitions and measurements, but for the scope of this paper, we simply measured the time in milliseconds it took the server to respond to each request. This is the main metric that was measured in the experiment.

4. **Costs** are the dollar equivalent of the computing resources that is used for the servers. For the Serverless architecture, the user is charged for each function invocation and we will take an average of per invocation cost from some popular cloud providers. For traditional servers, usually costs are determined by the hour of occupying the resource.

Before moving onto the comparison, it will be helpful to learn (or refresh ourselves) about the basics of web servers. In the background section, how serverless and traditional servers handle requests is explained which is a vital concept for the rest of the paper.

Background

Web servers are powering the software we use from mobile applications to websites to ATM machines we use everyday, but what exactly are they and why do we need them? To start with the simpler question, let's examine what they are. Web servers are computers that are dedicated to handle requests sent from clients through the internet. Clients in this context can be websites, mobile applications or ATM machines, simply any device that can communicate via the internet. Secondly, for the why, we need web servers to bridge the gap between different clients. All clients speak to the server which is the centralized single source of truth for the data. When an ATM machine reads your debit card, it sends a request to the server to authenticate, or ask, who you are, or check if you have enough funds that you are requesting to withdraw. Without a centralized server that persists data, accessing this information would not be possible since only the ATM that you deposited cash would know that information and this wouldn't be helpful when you try to use another ATM. Figure 1 demonstrates a simple request and a response mechanism that is widely used across web and mobile applications.

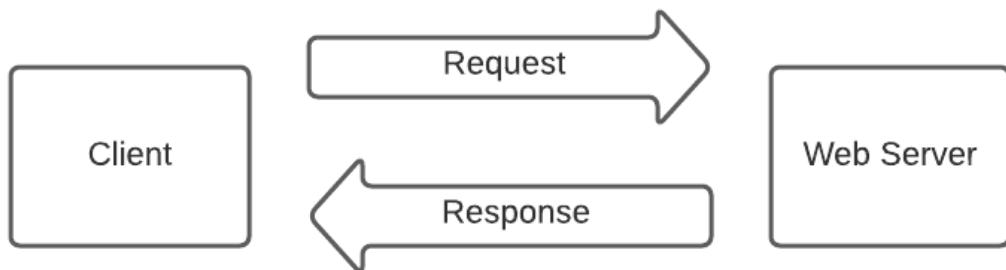


Figure 1: Client and Server Interaction

Now that we have a brief understanding of web servers it's time to explore how these computers are set up. It is fairly straightforward that no one would want their personal computer

to sit plugged into a power source handling web servers all day, because they wouldn't be able to use it! Buying a separate computer to do just that was the only possible way 20 or more years ago. This introduced a huge entry barrier for student projects or small scale businesses who needed web servers to run simply because no one could afford a few hundred dollar computer just to power their small project. Moreover, setting up the infrastructure to handle all these requests from all over the world in a **reliable** manner was a job that required many senior engineers to collaborate. If your project got popular and was getting requests that your single web server couldn't handle on its own you would come across all these **scalability** issues which would distract you from building your product and deal with the infrastructure problems.

Luckily, in the early 2000's the cloud revolution began taking place and tackled removing the barrier for managing web servers from the comfort of your desk. Cloud simply enables developers to rent computers from big cloud providers such as Amazon, Microsoft and Google and pay only for the computing they use. This approach is cost effective, and tens of thousands of world's best engineers are making sure that these services are reliable. It is a best of both worlds situation for small engineering teams.

Although the process of managing web servers was greatly simplified, it still presented great problems for development teams such as scalability and performance. Now everyone could spin up a server in Amazon data centers all over the world, but the users were expecting very fast and reliable web and mobile experiences. This meant that the servers had to respond to requests within a few hundred milliseconds and do this 24/7. Soon, some engineers specialized in managing the infrastructure of servers within cloud services which was becoming another loop of problems for small engineering teams. This is what 'serverless' aimed to tackle in the past few years.

Before exploring what serverless is, however, understanding how **traditional servers** are working will help comparing the two. Cloud providers allow users to easily rent a computer (server) that is dedicated for them. This means that you are occupying that computer all the time, which means that other users cannot utilize it. Due to scarcity, you are asked to pay for each second you are occupying that computer even if your web server is not receiving any requests from clients. It is not so hard to see that this is not the most efficient way to go, since you are paying for unutilized resources which is not optimal for a small business trying to cut all unnecessary costs. On top of efficiency, your traditional server has limited computing capabilities. When you are spinning up one, you choose the CPU and RAM resources allocated to that instance. This means that if your application gets more popular and your server is not able to handle some of the requests due to lack of computing power, you as the engineer have to step in and either increase the computing resources of the instance, or spin up more servers. I am not even mentioning a load balancer that needs to be configured to redirect incoming requests to the multiple instances in an efficient way. It is obvious that although spinning up traditional servers is an easy process, there are a lot of problems to arise in the future.

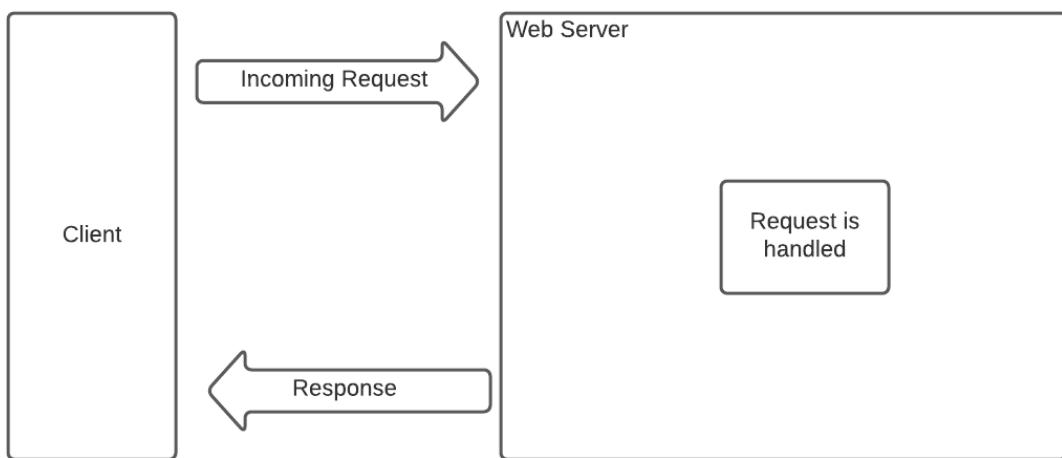


Figure 2: Traditional servers request handling

Serverless, as the name suggests, does not actually mean that it is server-less and there are no servers. There are still servers that are used to handle incoming requests. The ‘less’ part refers to the fact that the developer is not the one who is managing this server; but instead the serverless framework is responsible for orchestrating the servers that run your code. To be more exact, let’s discuss the lifecycle of how the serverless framework is orchestrating the servers for you.

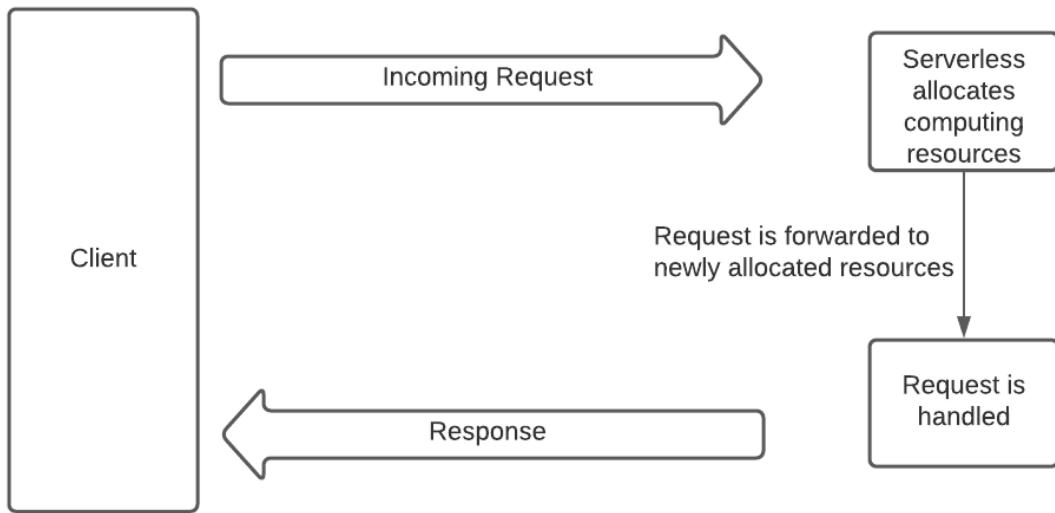


Figure 3: Serverless request handling

An incoming network request is detected and serverless allocates a computing resource, or reuses a previously allocated resource. After this step, it is similar to the traditional servers in terms of handling the request and returning a response.

The Experiment

The experiment consists of three main components to mock the real life use case of a web application. Source code that is used to conduct the experiment can be found at github.com/egecavusoglu/sls-experiment. The three actors are:

- The client,
- The serverless web server,
- The traditional web server.

The client is responsible for sending requests to both of the servers, collecting the requests and saving these transactions to a database.

A transaction between the web servers takes around a few hundred milliseconds, which is a quick process. To amplify the time it takes to complete a transaction, the serverless and the traditional web servers were both placed in data servers in the East Coast; and the client in Frankfurt, Germany so that each request has to travel across the ocean which increases the response times. It is also important to note that all of the computing resources were hosted on Amazon Web Services (AWS), since it's the leading vendor in the cloud provider industry. Traditional servers were hosted in **EC2 t2.micro** instances and serverless functions were hosted in **Lambda (1024MB)** configuration.

The Client

The client was scheduled to send 5 consecutive requests to each web server every hour for about 48 hours and generated **270 data points**. Since we are mocking a small scale project,

an individual user interacting with the application every hour and making a handful of requests seemed like a reasonable estimation.

Web Servers

For both of the web servers, the internal request handling logic was controlled so that they handled requests the same way. This also ensures that the time complexity of the work that is done in each request is unified. Let's take a look at Figure 4 to understand what the servers are doing.

```
function generateUuid() {
    return uuid.v1();
}

app.get("/", async (req, res) => {
    res.status(200).json({
        isSuccess: true,
        data: {
            uuid: generateUuid(),
        },
    });
});
```

Figure 5: Web server source code.

We are simply generating a unique identifier for the response, and sending it back to the client. Other actions such as updating a database, sending an email or processing a large request payload are excluded on purpose to just compare the raw performance of the servers. It is important to note that your web application will most likely be doing these kinds of things which will impact performance.

Database Schema

For the client to record every transaction to the database, a schema has to be declared as seen on Figure 5. The 3 important fields in the schema that are relevant for the experiment are:

- **service** has two possible values. “lambda” for serverless function and “traditional” for the traditional server. It is used to differentiate which server sent the response.
- **execution_time_ms** is the duration to execute the request and return a response measured from the client. It’s a numerical value in milliseconds.
- **execution_result** denotes if the request was successfully handled or failed. Two possible values are “success” and “failure” which are self explanatory.

```
const TransactionSchema = mongoose.Schema({
  _id: mongoose.Schema.Types.ObjectId,
  created_at: { type: Date, default: Date.now },
  service: {
    type: String,
    enum: ["lambda", "traditional"],
  },
  execution_time_ms: {
    type: Number,
  },
  execution_result: {
    type: String,
    enum: ["success", "failure"],
  },
  response_uid: {
    type: String,
  },
});
```

Figure 5: Transaction schema

Other metadata associated with each transaction was also recorded like the unique identifier, creation date, and the response unique identifier. These are not relevant to the experiment results so there is no need to dive deeper into them.

Results

There are 2 graphs that show us relevant data from the experiment. First one is the average response time per service, and the second one is the request execution status vs service.

In Figure 6, we can see the average time to handle requests by the serverless (lambda) and the traditional server. We can see that traditional servers are over twice as fast at ~183.8ms for the mean time to handle a request, whereas for the lambda it is ~388.2ms. To be exact,

$$\text{traditional servers are } \frac{388.2 - 183.8}{183.8} = 1.11 = 111\% \text{ faster.}$$

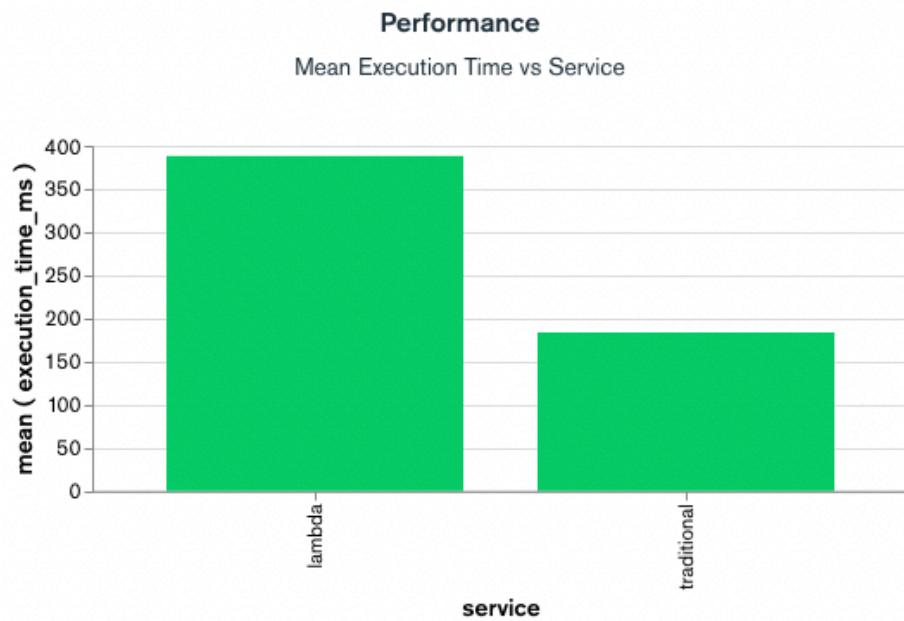


Figure 6

In Figure 7, we can see that reliability is very consistent for both of the services. This is expected since AWS is known to be a very reliable cloud provider. It is also crucial to have healthy services so it's reassuring to see both services succeed in all 135 requests sent to each of them.



Figure 7

Analysis

Now that we have the results from the experiment, we can start the comparison with respect to the criteria introduced in the beginning.

Scalability

Serverless functions have a fixed lifespan. When an incoming request is received, a computing resource is allocated, function is executed, and terminated. This means that they are stateless, and independent of each other. We could run thousands or even millions of functions at the same time and none of those would be interfering with one another. This is the strongest perk of the serverless architecture, the scalability is almost limitless. Never worrying about scalability when building services for a product enhances developer experience, thus is a big plus for the serverless.

On the other hand, traditional servers have limited computing capabilities determined when they are initialized. This means that either a person, which is a very bad idea, or another service has to track the usage and manipulate the computing resources with respect to the load of the incoming requests. It is definitely possible to manage scalability with configuration options, and all the cloud providers have robust services to help you with it. Still, they do not come out of the box and they have to be taken into account by the developers.

Reliability

Reliability is the single most important metric for a web service. It simply means that the service is healthy and able to accomplish the request as expected. When reliability is low, users

will think that the application is broken, or not trustworthy. This also means that you are not able to provide the services you promise to. This is why cloud providers are very careful about their reliability measures. Usually reliability is measured by the number of nines, like “five nines”, “six nines” and so on which can be better contextualized in Figure 9.

Availability %	Downtime per year ^[note 1]	Downtime per quarter	Downtime per month	Downtime per week	Downtime per day (24 hours)
90% ("one nine")	36.53 days	9.13 days	73.05 hours	16.80 hours	2.40 hours
95% ("one and a half nines")	18.26 days	4.56 days	36.53 hours	8.40 hours	1.20 hours
97%	10.96 days	2.74 days	21.92 hours	5.04 hours	43.20 minutes
98%	7.31 days	43.86 hours	14.61 hours	3.36 hours	28.80 minutes
99% ("two nines")	3.65 days	21.9 hours	7.31 hours	1.68 hours	14.40 minutes
99.5% ("two and a half nines")	1.83 days	10.98 hours	3.65 hours	50.40 minutes	7.20 minutes
99.8%	17.53 hours	4.38 hours	87.66 minutes	20.16 minutes	2.88 minutes
99.9% ("three nines")	8.77 hours	2.19 hours	43.83 minutes	10.08 minutes	1.44 minutes
99.95% ("three and a half nines")	4.38 hours	65.7 minutes	21.92 minutes	5.04 minutes	43.20 seconds
99.99% ("four nines")	52.60 minutes	13.15 minutes	4.38 minutes	1.01 minutes	8.64 seconds
99.995% ("four and a half nines")	26.30 minutes	6.57 minutes	2.19 minutes	30.24 seconds	4.32 seconds
99.999% ("five nines")	5.26 minutes	1.31 minutes	26.30 seconds	6.05 seconds	864.00 milliseconds
99.9999% ("six nines")	31.56 seconds	7.89 seconds	2.63 seconds	604.80 milliseconds	86.40 milliseconds
99.99999% ("seven nines")	3.16 seconds	0.79 seconds	262.98 milliseconds	60.48 milliseconds	8.64 milliseconds
99.999999% ("eight nines")	315.58 milliseconds	78.89 milliseconds	26.30 milliseconds	6.05 milliseconds	864.00 microseconds
99.9999999% ("nine nines")	31.56 milliseconds	7.89 milliseconds	2.63 milliseconds	604.80 microseconds	86.40 microseconds

Figure 9: From Wikipedia, High availability [6]

In the experiment we saw that all 270 requests to both servers were executed successfully which was reassuring.

It's also important to note that, traditional servers may crash occasionally, and unless the developer sets up necessary mechanisms to rerun the service, it will be down even if the underlying computing resource is healthy. Lambdas, due to their lifecycle managed by the serverless architecture, are much more easy to manage which is another big plus for the developer experience and are less error prone.

Performance

Performance is a metric that is very important to the end user. When a competitor app is loading in milliseconds, your application cannot afford to take seconds to load. Therefore, serverless was built with this in mind. However, by nature, it has to pay the fixed cost of allocating computing resources upon the receiving of a request, which is called a “cold start”. It is also important to note that serverless functions are smart enough to reuse some computing resources when multiple requests are received in short intervals. So if there are back to back requests to your function, it will reuse the computing resource allocated for the first one and save time wasted on cold start. There are also other ways of keeping your serverless functions “warm”, by sending them arbitrary requests in a schedule so that they always remain warm. Please note, that every request sent is subject to a charge by the cloud provider so it is important to take this approach into consideration.

Traditional servers however are listening to requests in their dedicated virtual machine all the time, so unless the current computing resources are exhausted due to the high load of incoming requests, they will handle the request right away. This is the fastest possible approach in cloud computing that we know of, so there is not much to discuss.

Looking at the experiment data, we see that traditional servers performed more than twice as fast than serverless functions. This was expected, by nature traditional servers are faster. Let's contextualize the data for better reference. Note that both 183 and 388ms are still very fast for waiting for a request to be executed and hardly differentiable by the experience we see on the screen. Moreover, according to littledata.io [2], average response time of the 6,500 servers that were surveyed was around 493 ms. Top 10% of the servers were able to respond in 206ms, and the bottom 20% did 849 ms or worse. Hrank, a hosting service provider, also states “200-350ms

is considered fast, 400-700ms is average, and all the rest can be called slow.” [3] So, the performance of lambdas are very acceptable and fall into the faster end of the spectrum.

Cost

One of the most important metrics when running a business is profitability. So keeping costs low and efficient is always an important factor. This is another aspect where the serverless shines. Traditional server pricing is pretty simple, you pay for the hours you use the service. Since for a reliable service they must run all the time, your cost is pretty much fixed if scaling is not taken into account. In Serverless functions however, the pricing is a pay-as-you-go model. What this means is that, if for a few days your function is not called by anyone, you pay \$0. When the service is actually used, you are charged for the number of invocations and the duration it takes to execute the function. This is the ideal scenario for businesses, you are not subject to fixed costs and if your service is used a lot, then you are making more money so paying more for the computing resources becomes easier.

To compare the cost in the experiment, let's calculate the monthly cost of using each service.

For traditional servers, the price per hour of a t2.micro instance is \$0.0116 [4], and there are 730 hours in a month. So the monthly cost becomes $\$0.0116 * 730 \approx \8.50

For lambdas, we are invoking them 5 times every hour for a month. Total invocation equals $5 * 24 * 30 = 3600$ and the cost is \$0.20 per 1 million requests. We also know that average execution time is ~388ms and price per ms is \$0.000000167 for 1024 MB lambda configuration. [5] So the total becomes

$\$0.20 (since 3600 < 1M) + 3600 * 388 * \$0.000000167 \approx \$0.22$. Even if it was

invoked 1M times, the price would go up to around \$6.50. So it's safe to say that they are cost efficient.

Conclusion

The serverless framework, and managing web applications with them has been a breakthrough in cloud computing over the recent years. It is for sure that traditional servers are not going anywhere, and they will keep powering the enormous services that are used by billions everyday. For the scope of small businesses and startups, however, serverless removes the hassle of so many infrastructural efforts. Having infinite scalability and high reliability are not priorities a startup can invest their engineering resources to, so having them out of the box is very powerful. The small setback in the performance is not significant, and is definitely tolerable for small scale businesses. Finally in terms of costs, the serverless offers the perfect model with the pay as you go approach. To sum up, building web applications with serverless looks like the best option for small to midsize startups to focus more on their business and product instead of the infrastructure.

References

1. “Cloud scalability,” *VMware*. [Online]. Available:
<https://www.vmware.com/topics/glossary/content/cloud-scalability#:~:text=Cloud%20scalability%20in%20cloud%20computing,its%20exploding%20popularity%20with%20businesses>.
[Accessed: 13-Dec-2021].
2. “What is the average server response time?,” *Littledata*. [Online]. Available:
<https://www.littledata.io/average/server-response-time>. [Accessed: 13-Dec-2021].
3. “Why server response time is important,” *HRank.com*. [Online]. Available:
<https://www.hrank.com/why-server-response-time-is-important#:~:text=What%20is%20average%20server%20response,more%20than%20800ms%20is%20slow>. [Accessed: 13-Dec-2021].
4. D. J. Daly and D. J. Daly, “Economics 2: EC2,” *Amazon*, 1987. [Online]. Available:
<https://aws.amazon.com/ec2/instance-types/t2/>. [Accessed: 13-Dec-2021].
5. D. A. V. I. D. MUSGRAVE, “Lambda,” *Amazon*, 2022. [Online]. Available:
<https://aws.amazon.com/lambda/pricing/>. [Accessed: 13-Dec-2021].
6. “High availability,” *Wikipedia*, 30-Nov-2021. [Online]. Available:
https://en.wikipedia.org/wiki/High_availability. [Accessed: 13-Dec-2021].